**Beginner’s Guide to Python:**

Table of Contents:

1. Installing Python
2. Lists
3. Loops
4. Regex
5. Working with files
6. Scraping
7. HTML Parsing with BeautifulSoup

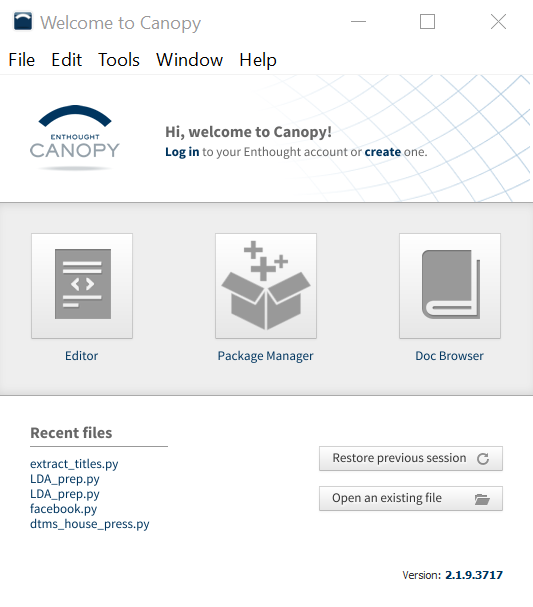
**Installing Python:**

To install Python and the Canopy environment, go to:

<https://store.enthought.com/downloads/>

Be sure to download the **Python 3.5** version appropriate for your os. Canopy is not strictly required for this class (if you already use a different environment, that’s fine), but it is **strongly recommended** for beginners to Python, since it contains a handy package manager.

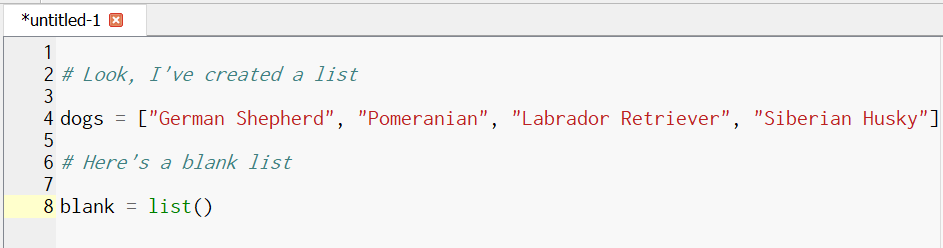
When you open up Canopy, you will see the following popup. Click on “Editor and then “Create a New File”.



**Lists in Python:**

**Creating Lists:**

R and Python are similar in many ways, but one of the biggest differences is the use of lists. While lists in Python are roughly analogous to vectors in R, they have several very different properties. Here, I will walk you through the basic properties of lists in Python.



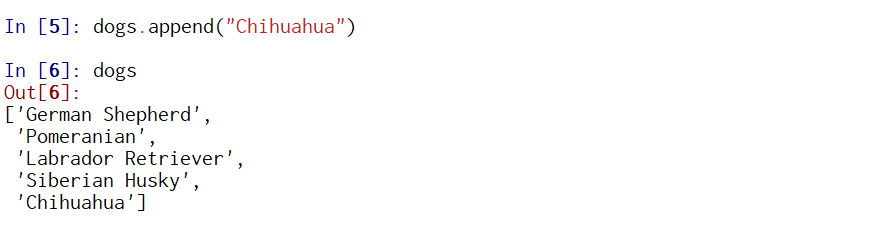
Unlike in R, lists in python are **zero-indexed**. This means that:



In R, dogs[1] would give you “German Shepherd”, while in Python (and most other languages), the first element of a list, array, etc is denoted by zero, not one.

**Adding to Lists:**

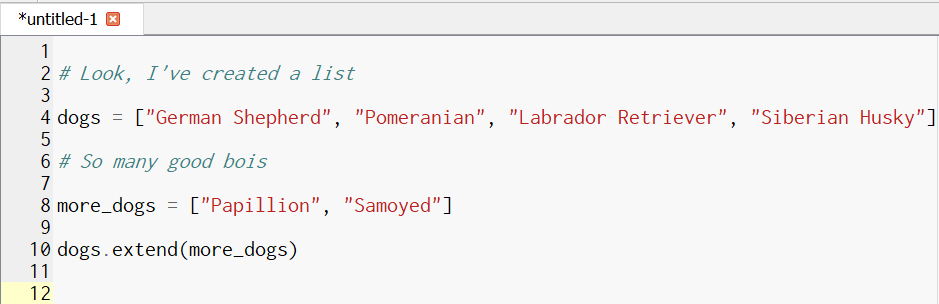
In Python, there are two functions to add to lists: **append()** and **extend()**



Here, we are adding the string “Chihuahua” to our list of dogs. The syntax is:

*list\_name.append(“Thing to append”)*

On the other hand, here’s the extend function:



Result:



Use the extend function to combine two lists, and the append function to add one item to a list.

**Important Note**:

Note that the syntax for append & extend is:

*list\_name.extend(new\_addition)*

NOT

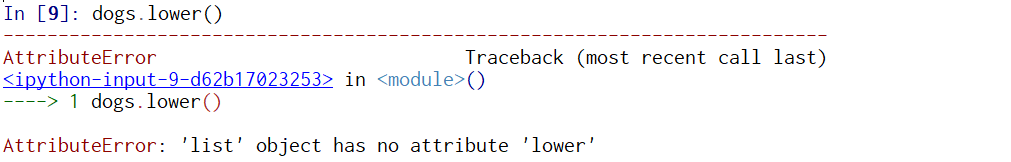
*list\_name = list\_name.extend(new\_addition)*

The append and extend functions in python automatically add the item onto the end of the list, even if you don’t overwrite the list. So be careful when testing code – you don’t want to add the same thing to the end of the list multiple times!

**List Comprehensions**

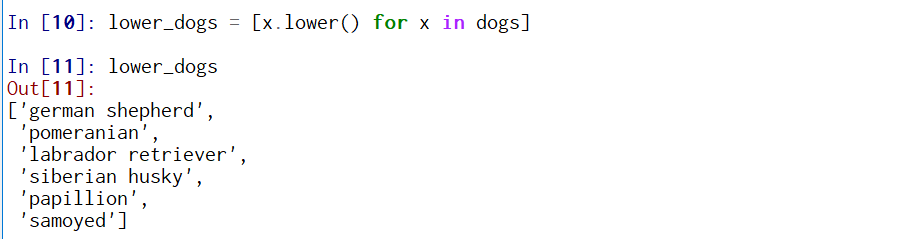
Let’s say you want to apply a function to your list. For example, you want to turn all your dog names lowercase.

In R, you’d just do something like *tolower(dogs)*. Let’s try the equivalent in Python:



Bummer! It doesn’t work. Lower() is a function that only works on strings, not on a list of strings.

Naturally, Python has a solution to this problem. It’s called a **list comprehension**:



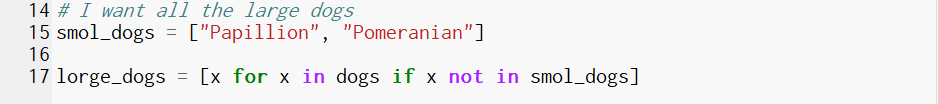
Here, we’re taking each item ***x*** of the list ***dogs*** and applying the ***lower()*** function to it, and then binding them all together as a new list, ***lower\_dogs***.

This code is roughly equivalent to the following for loop:

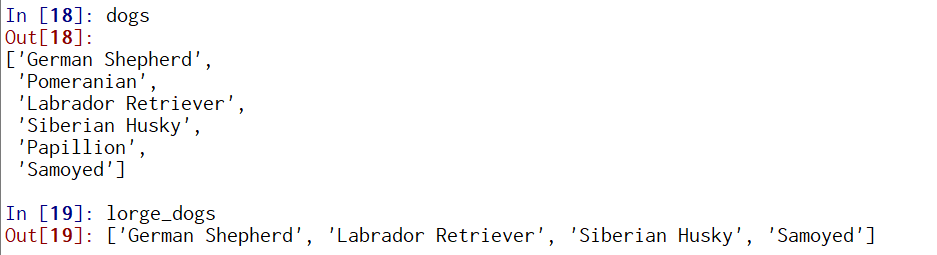


Most Python programmers use list comprehensions whenever possible, so even if you choose to use the loop version of this code, you need to be able to understand Python list comprehensions.

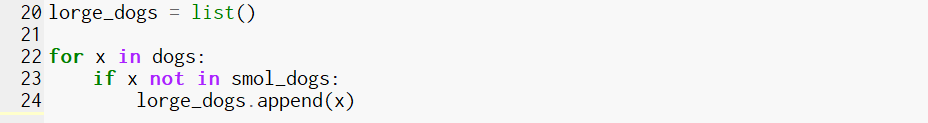
List comprehensions can be used for a variety of things, including filtering:



Here, we want the new list ***lorge\_dogs*** to contain all items ***x*** that are in the ***dogs*** list but not in the ***smol\_dogs*** list.

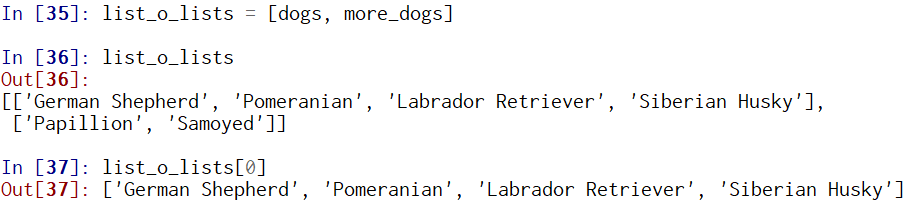
Result:

Again, this is the equivalent to the following for loop:



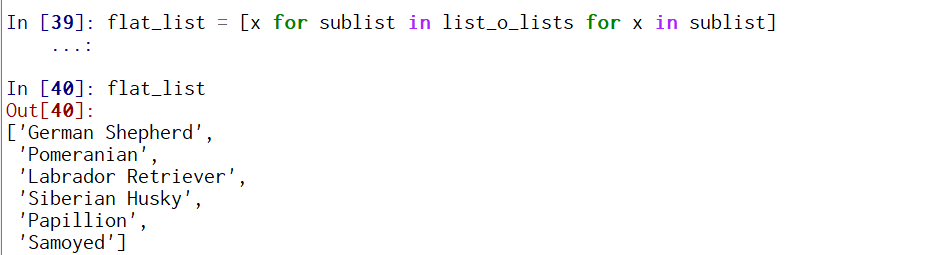
This is much less efficient than using the list comprehension.

Here’s a final list comprehension meant to get you out of the incredibly annoying jam known as the **list-of-lists**. Here’s what that is:



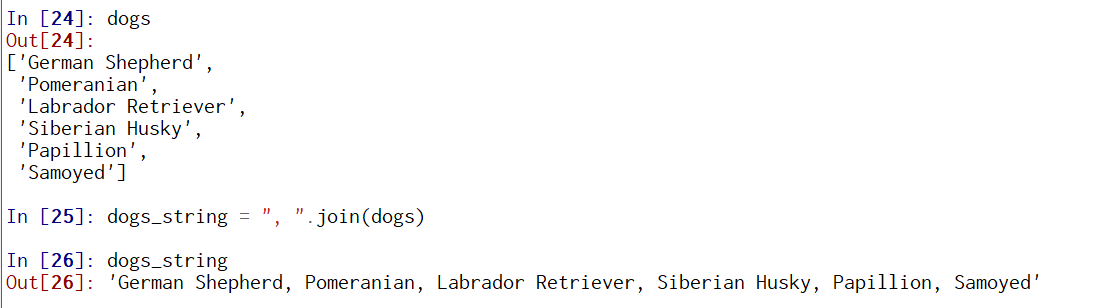
Now, in this case, I can easily re-run the code correctly using the extend function, but what if this were the result of hours of scraping? Now, I’m stuck with a list, each item of which is *also* a list (you can also create a list-of-lists-of-lists etc). There are some situations where lists-of-lists are intentional, but in most situations I’ve encountered, they’re unintentional and really annoying.

Here’s how to fix it:



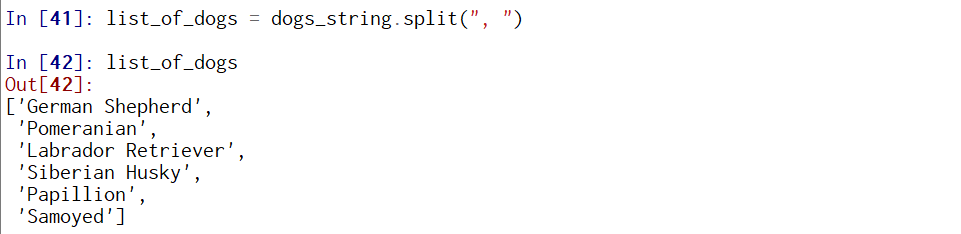
**Lists to Strings and Vice Versa:**

Let’s say I want to collapse a list to a string in Python. How would I do that?



Using the **join()** function is actually very similar to using append or extend. Here, we are taking all of the items from the list ***dogs*** and concatenating them with a **,** in between.

And the opposite:



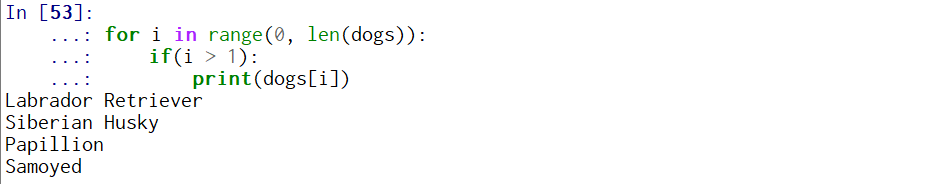
**Loops in Python:**

Loops in Python are similar to loops in R:



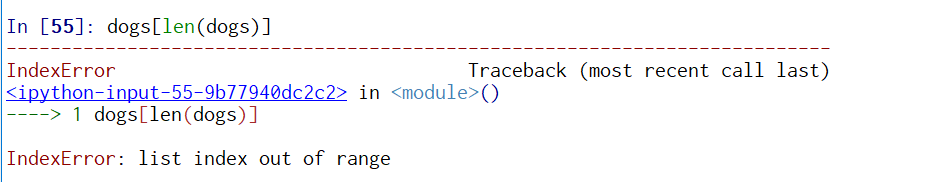
However, loops and if-else statements in Python use indents in the places where R uses “{“ and “}”.

In Python, you need to be aware of zero indexing when writing loops that iterate over indices in a list



Here, the index i can take the value from [0, 6) – eg values 0,1,2,3,4,5 (6 is not included). The loop prints the last 4 objects in the list, as the first two have an index of 0 and 1, which is not less than 1.

Zero indexing also explains the following error:



In R, if dogs were a vector, running *dogs[length(dogs)]* would give you the last element of the vector. Not so in Python! To get the last element of a Python list, you need to do *dogs[len(dogs)-1]*.

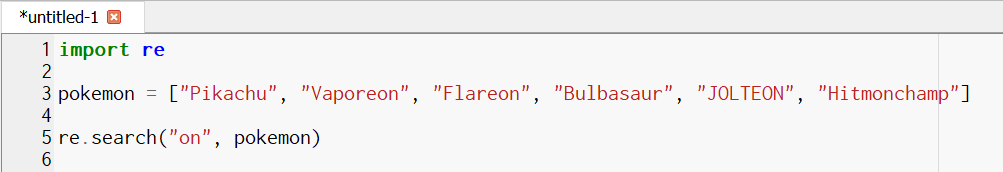
**Python Regular Expressions:**

Here, I will provide information on the use of regular expressions in Python.

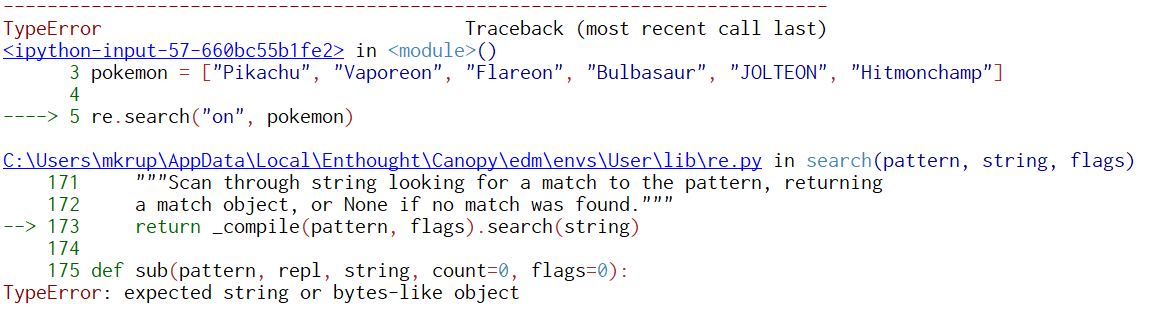
**re.search()**,**re.split()**, , and **re.sub()** are three of the most commonly used regular expression functions in Python. I’ll go through each of these in order.

To use regular expressions, you need to import the regular expression package as shown in the first line.

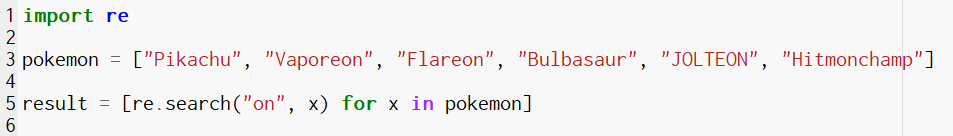
Say we want to extract the words “Vaporeon”, “Flareon”, and “JOLTEON” from this list. To do this, we use **re.search()**, and we notice that all three of them end in the letters “on”



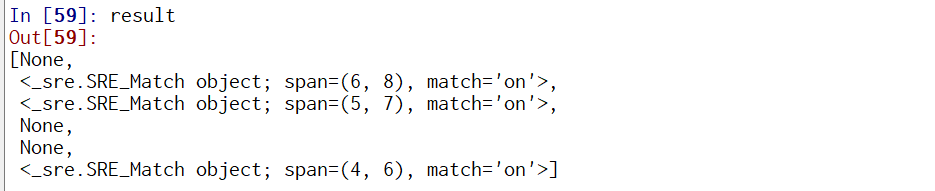
Result:



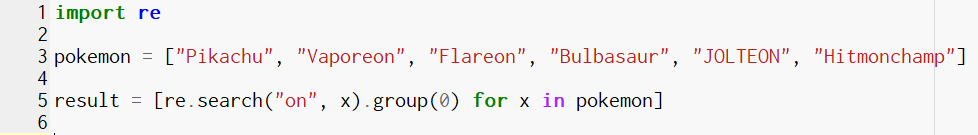
We get an error because we’re trying to use a regular expression on a list and the regular expression function only works on strings. To get around this, we use a list comprehension.



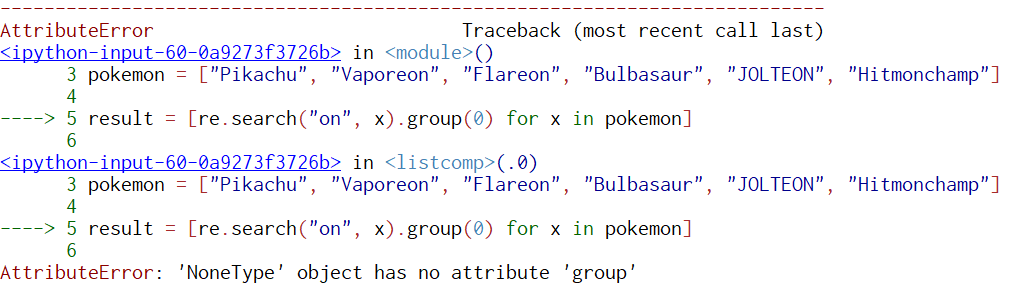
Here’s our result:



This is… not super helpful. That’s because in order to extract the actual the actual matches, we need to do **re.search(pattern, string).group(0)**

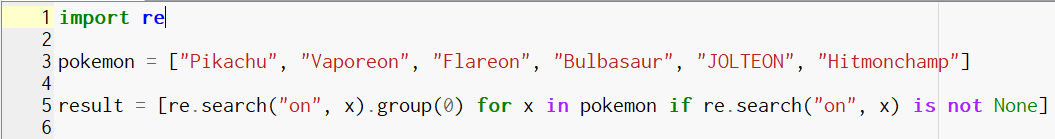
****

And as a result we get…



An error!

That’s because some of the strings in the list didn’t match “on”, and when Python tries to apply the group() function, if gives us an error. Thankfully, this is pretty easy to fix.

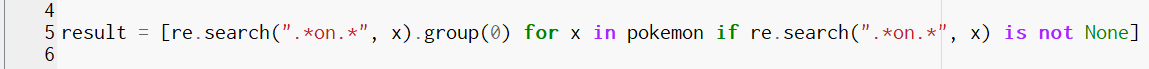


The only difference between this list comprehension and the previous one is that we have the *if re.search(“on”, x) is not None* at the end, which prevents Python from trying to apply the group() function to any null results.

Result:



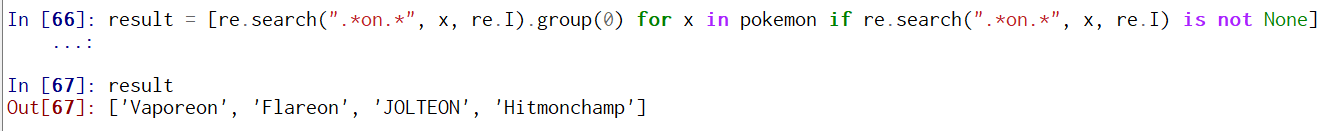
Hmm… still not what we’re looking for. To get that, we do the following:





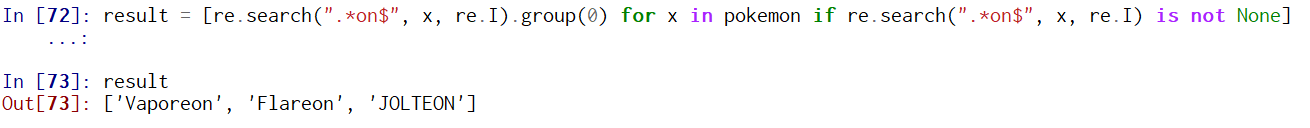
Closer! When we added the “.\*” to the regex on both ends, we are matching all of the characters on either side of the “on”. Still, our result is picking up “Hitmonchamp”, who we don’t want, and missing, “JOLTEON”, who we do.

To get the latter, we make our regression case-insensitive.



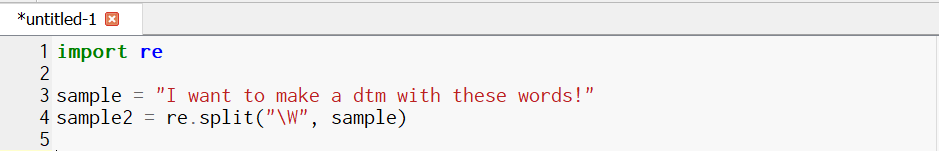
Adding **re.I** to the re.search() function makes the regex pick up upper and lower case letters.

Now, to get rid of Hitmonchamp, we can run the following:

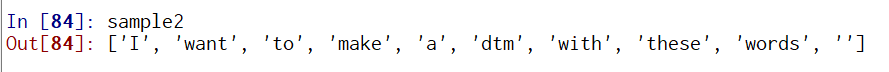


The $ matches the end of the string, which limits our results to strings that end in “on”.

Ok, now for **re.split()**



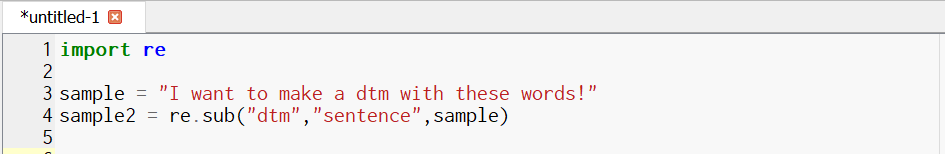
Result:

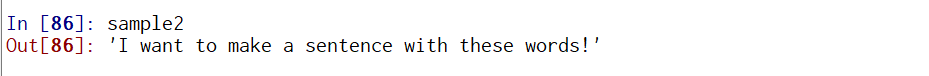


Simple enough! The reason we can use re.split() directly on ***sample*** without having to do a list comprehension is that ***sample*** is a **string, not a list**.

Finally, **re.sub()**:

Let’s say we want to replace the word “dtm” with the word “sentence”. We do this as follows:

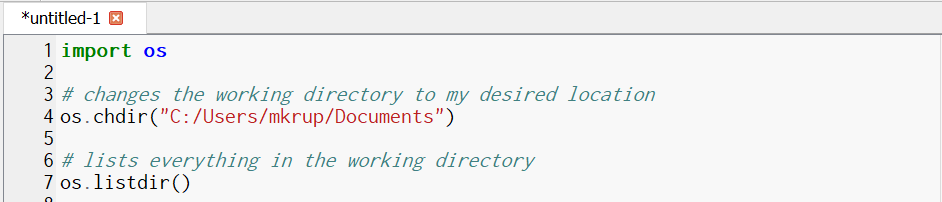




Simple enough! All of these functions can be combined with list comprehensions to process lists if necessary.

**Working with files in Python:**

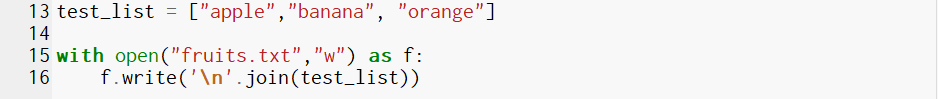
Here, I will discuss how to read and write files in Python.



Here’s how to read in a file in python:



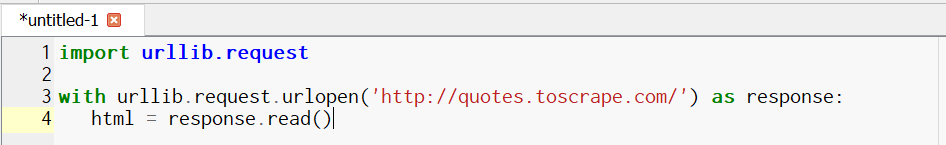
And (one way) how to write a list to a file in python:



Here, I write a list to a file in a way that creates line breaks between list items. If you wanted to put everything on the same line, you could use “ “ or “,” instead of “\n”

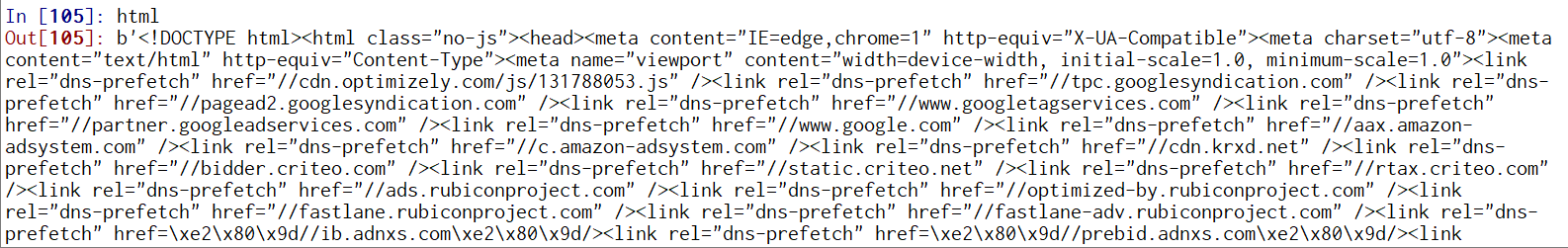
**Python Webscraping:**

There are many ways to scrape a webpage in Python. Today, we’ll use a simple one – the urllib package.



Here, we’re using the **urlopen()** function from the urllib package. Once urllib has called the webpage, we can get the text using the **read()** function.

Here it is:



We will talk about parsing this output in the next section.

This is all good and simple, but what happens when we want to download more than one webpage?

Let’s say we want to get all the text from the first 5 pages of this website. When we do a little sleuthing, we can tell that the pages in this website have urls that follow this pattern:

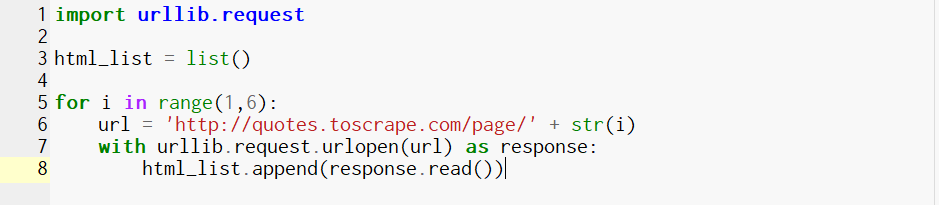
<http://quotes.toscrape.com/page/5/>

We can use a for loop to automatically iterate over the pages we’re interested in:



Here, in each iteration, we create a string object ***url***, with the iterator i representing each page we want to scrape. Then, we then feed the url into the urlopen() function.

This is all good and well, but each time we’re iterating through the loop, we’re overwriting the html from the previous page. We can deal with this the same way that we do in R – by capturing it in a list.

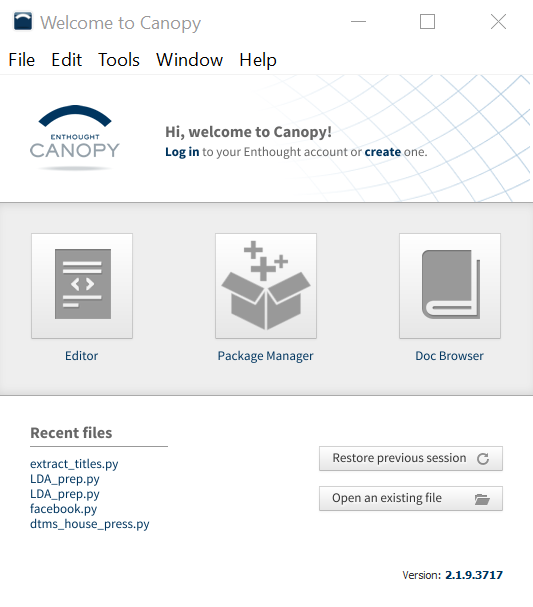


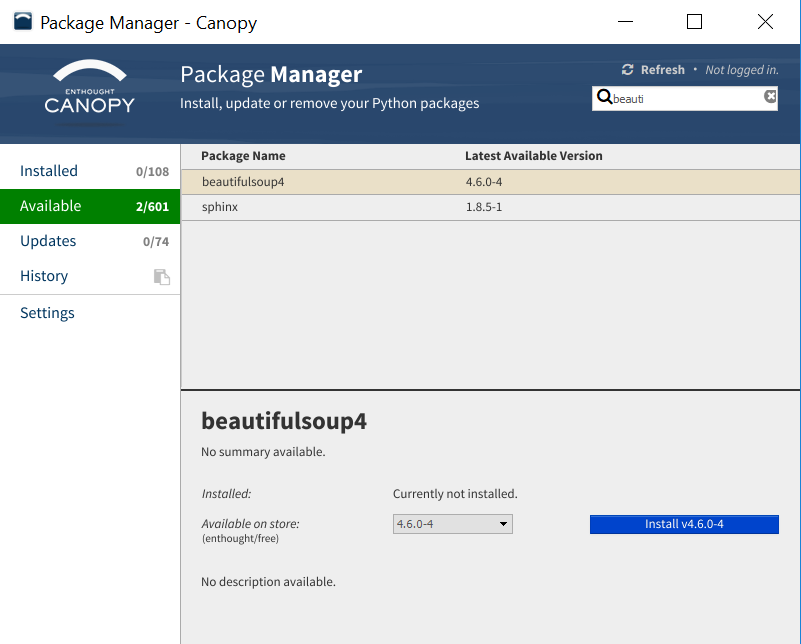
As a result, we have a simple scraper that iterates through the first 5 pages of this website and stores the html from each page in a list.

**Parsing HTML:**

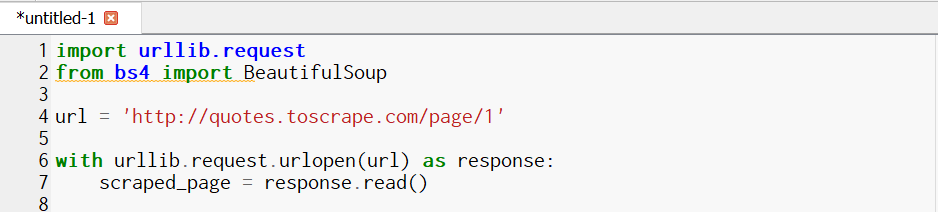
We’re going to use BeautifulSoup in order to clean up and grab info out of our scraped htmls.

To install BeautifulSoup, go to the popup you first got when starting canopy, and click on Package Manager and search for BeautifulSoup in the upper righthand corner.

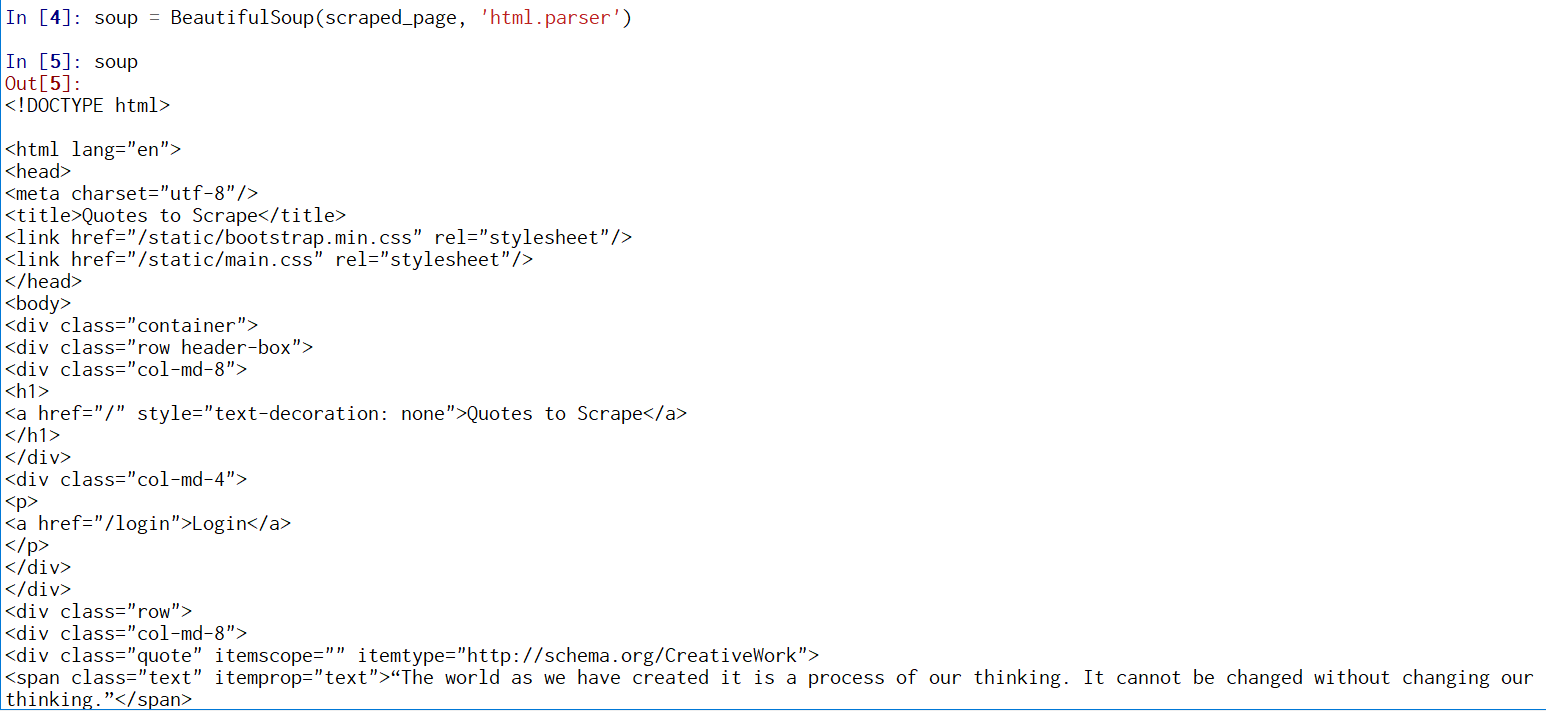




Now, let’s import BeautifulSoup and grab a page to parse:



Next, we actually parse the page using BeautifulSoup. It looks much better:



Now, you need to think about what exact data you want to get from this page:

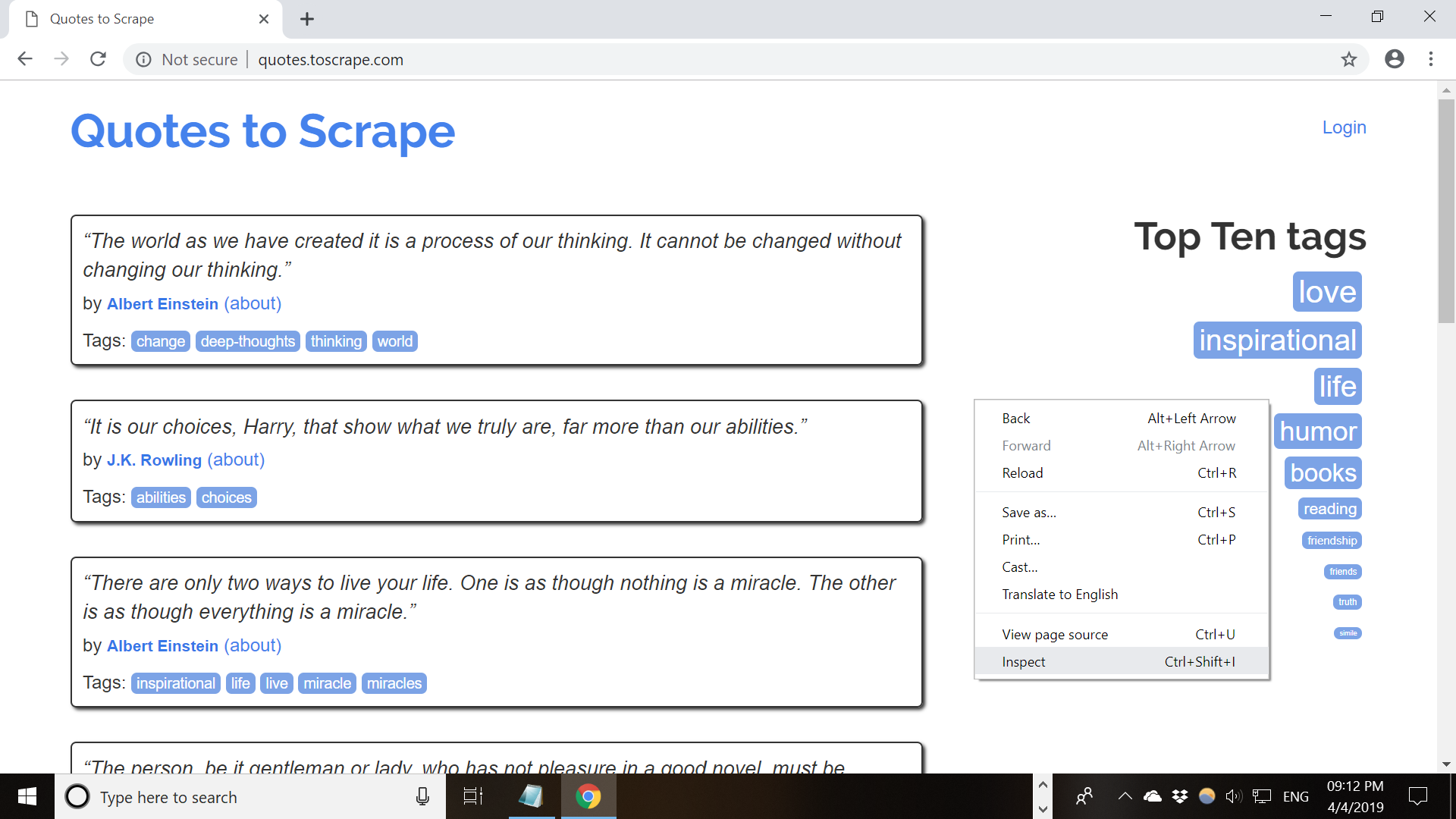
Do you want the text of the quotes?

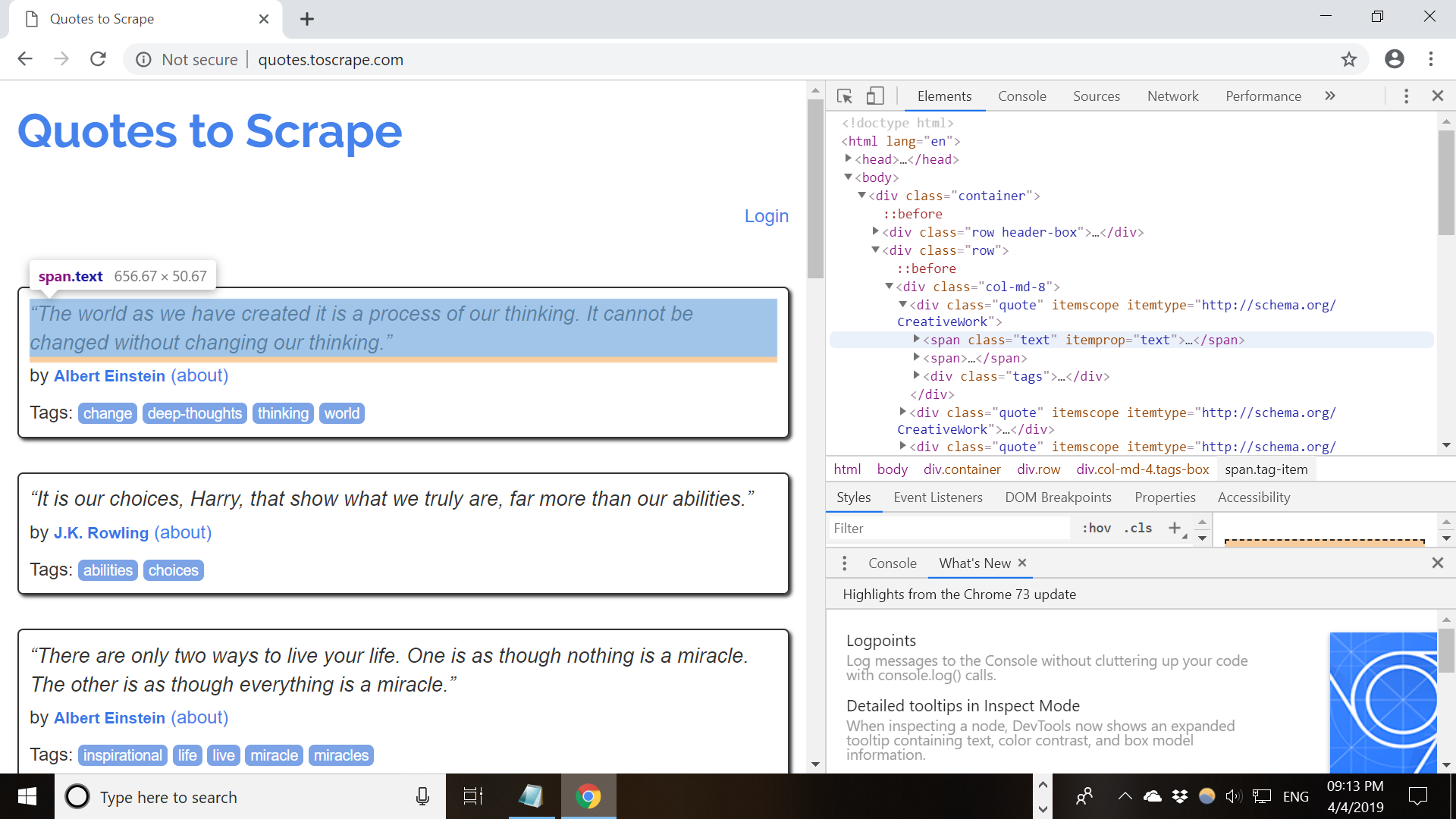
The authors?

The tags?

All of the above?

Before writing any further code, I like to use the Inspect (on Chrome) or Inspect Element (Firefox) to explore the webpage and figure out what I want to parse (as shown on the next page). This method allows you to scroll through the code, and the browser will highlight different parts of the webpage. It’s especially useful for more complex webpages.

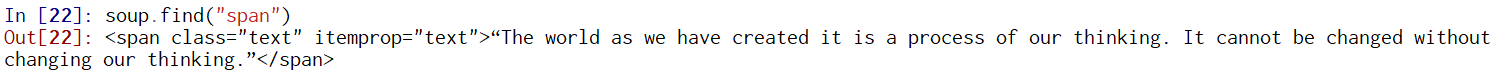




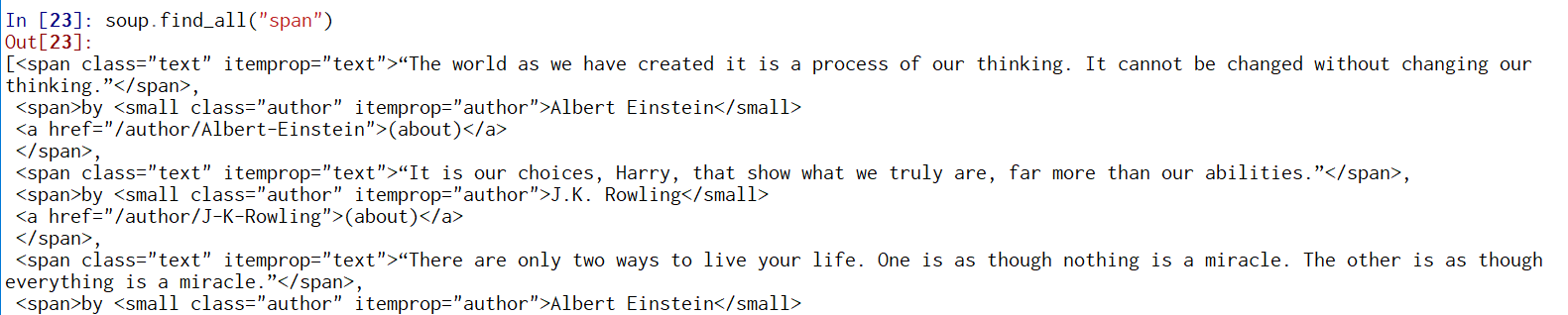
I decide I want to create two lists – one with the quote text, and one with the authors of each quote.

I am going to go through only a few of the functions from BeautifulSoup – the rest is in the very solid documentation you can find here:

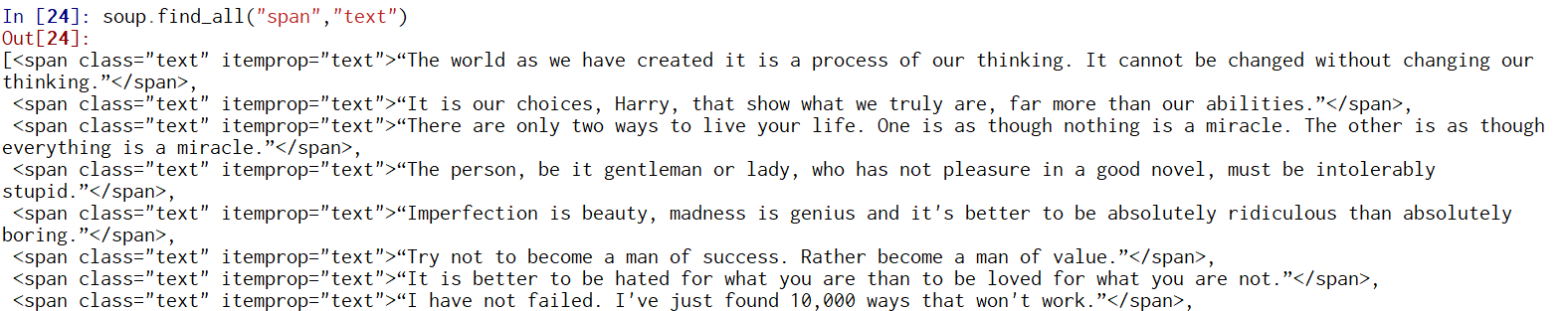
<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

As shown above, each quote is within a <span> tag. Maybe I can get all the quotes by using the span tag? 

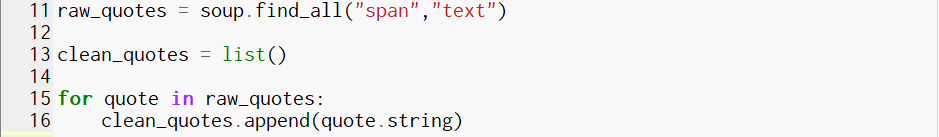
This looks pretty great, but it’s only one quote, and I want all the quotes on the page.

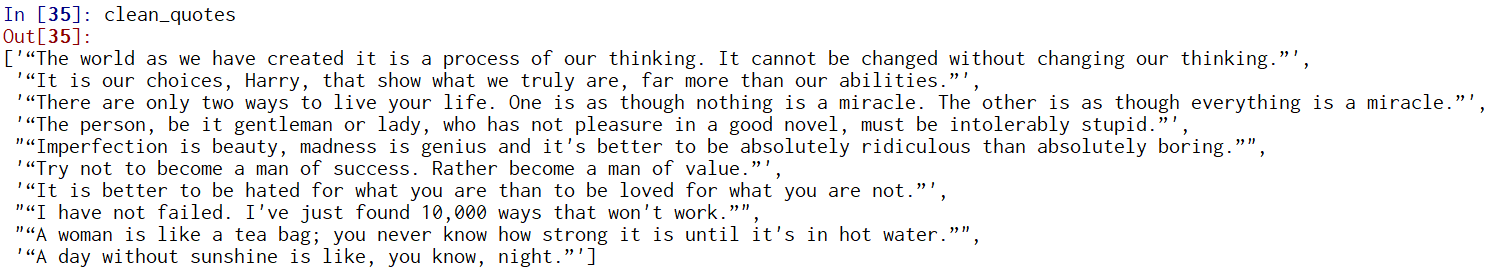


Hmm, here I do get all the quotes, but I also get a bunch of stuff I don’t want in this specific list. I notice that each quote is within a <span> tag that also has class=”text”



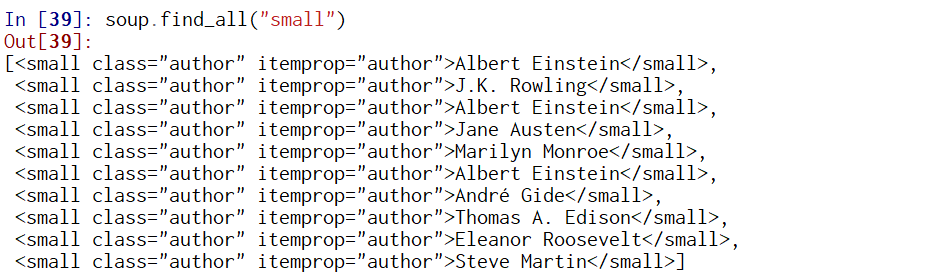
Nice! Now I just need to extract the text only. The string function grabs the text from an object created by BeautifulSoup



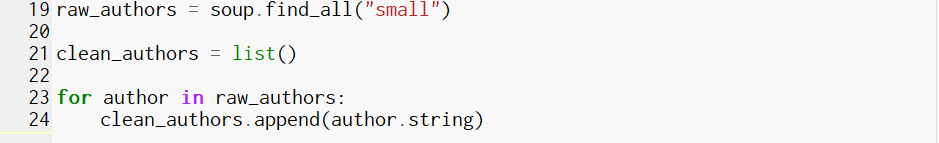


Now, I want to get the authors:

As noticed earlier, authors are in the <small> tag:

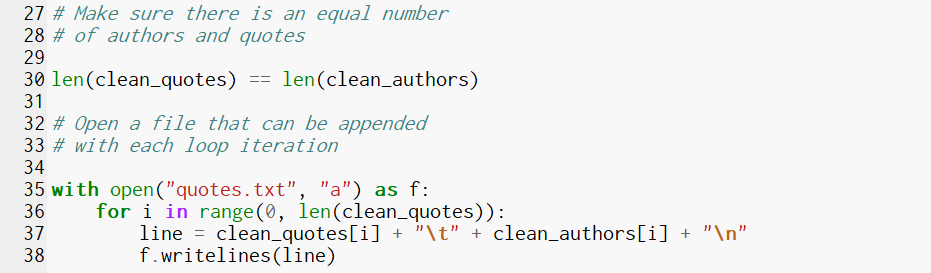


Now to get the names only…





Ok, now we want to write these nice lists to a text file:



This creates a file called “quotes.txt”. The permissions for this file are “a” , which means it can be appended to (eg, each iteration of the loop adds another line, instead of overwriting the previous one).

Each quote-author pair is separated by a tab (“\t”), and each quote-author par takes up one row (“\n”)